Program #3
Debugging
Due in Class Wednesday, November 26, 2003

Name: _____

Lab Time: _____

Grade: _____/30

A Car Dealership Database

For this program you will be debugging a car dealership database program. Below is the specification for this program. A correct executable named `Dealer-okay` exists in the directory `~csci3308/arch/$ARCH/bin` for all architectures. The correct executable is for you to understand how the program is supposed to work. We are also providing the correct executable as part of the program specification for this assignment. You can use the Dealer-okay program to generate the expected output for the test case that ships with the buggy version of the system. The buggy source code and a test case are contained in a tar file `~csci3308/src/Dealer-bugs.tar`. This is the source code you will debug.

Unpack the source code into your source directory. Before you compile the program, make sure you are using the g++ compiler in </usr/bin> or the one in </tools/cs/gcc-3.3/bin/g++>. Now go to the architecture-specific build directory for Dealer-bugs (note: you will need to create this directory first!) and build the program (you can ignore the warnings about "deprecated" headers):

```
make -f ~/csci3308/src/Dealer-bugs/Makefile install
```

Type rehash and verify that Dealer-bugs is in your path.

You will now want to have two terminal windows open. One should stay in the architecture-specific build directory and the other should stay in the Dealer-bugs src directory. You will run test cases and make fixes to the Dealer-bugs source code in the src directory. You will build and debug the program in the build directory. (Copy the easytest.in file from the src directory into the build directory, so you have it in both places.) Each time you fix a bug in the src directory, you will re-run the above make command to build Dealer-bugs to see if the bug has been fixed. Feel free to write new test cases for this program. Indeed, the supplied test case can be viewed as an "acceptance test" for the Dealer-bugs program, and as such, you may find it more productive to create smaller test cases that test specific functions of the program. If you do so, you can use the Dealer-okay program to generate the expected output for the test case, as mentioned above.

In the src directory, run the correct version of the dealer program on the test case that came with the Dealer-bugs distribution. Dealer-okay is located in the csci3308 bin directory so it should be in your path:

```
Dealer-okay < easytest.in
```

Now that you know the expected output, run the buggy version you just compiled:

```
Dealer-bugs < easytest.in
```

There are seven bugs in the program. You must debug the program until the easytest.in test case passes. Be sure to use the version of gdb in </usr/bin>.

When you find a bug, use RCS version control to track changes to the source files. Check in the original buggy files, and check in each file when you make a change. When you are done, use the program `rcsdiff` to print out the differences between your correct version, and the original buggy files. The only thing you will turn in is the printout of this `rcsdiff`. From this printout we will see whether you found all the bugs and fixed them correctly. Please include a comment in the code about what you did to fix each bug and make sure to include your name and lab section on the printout.

This program is due in lecture on Wednesday, November 26th.

Requirements Specification of a Car Dealership Database

1. The database stores information about cars. Information about a car is entered into the database when a car is added to the inventory of the dealership. It is removed from the database when the car is sold. The dealer can ask to print information about a particular car given its identification number. The dealer can also ask for a printout of all the cars in the inventory in three different ways

2. The program reads a sequence of commands from `stdin`. The input should be formatted as a sequence of integers separated by whitespace. The first integer selects the command to perform. Some commands take a certain number of parameters. If the command takes parameters then one integer is read from stdin for each parameter. After the command is executed the process repeats with the next integer being interpreted as a command until all input is exhausted.

3. Splitting the input across multiple lines has no effect except that a newline is considered whitespace, so there can be multiple commands on a single line, or a single command can be split across multiple lines.

4. The commands are as follows

   1 Add a car. Has 4 parameters: CarId, Speed, Doors, Price.

   2 Remove a car. Has 1 parameter: CarId.

   3 Print information about a particular car. Has 1 parameter: CarId.

   4 Print information about all cars ordered by identification number, from smallest to largest. Has no parameters.

   5 Print information about all cars ordered by time the car has been in inventory, longest to shortest. Has no parameters.

   6 Print information about all cars ordered by time in inventory, shortest to longest. Has no parameters.

5. Output should only occur in response to commands 3 through 6. For each car that is printed the program should print the CarId, Speed, Doors, and Price all on one line. Each car should be printed on a separate line even if subsequent output is the result of the next command.

6. The following error conditions must be checked by the database program.

   - A command number less than zero.
   - Adding a car whose identification number is already in the database.
   - Selling or asking for information about a car whose identification number is not in the database.

   The response to an error should be that the current command is ignored. The program can assume that the number of parameters provided for a command will always be correct.